(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
3 May 2001 (03.05.2001)

PCT

(10) International Publication Number
WO 01/31439 A2

(51) International Patent Classification[7]: G06F 9/44

(21) International Application Number: PCT/US00/26785

(22) International Filing Date:
28 September 2000 (28.09.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/412,114 10 October 1999 (10.10.1999) US

(71) Applicant: PLATINUM TECHNOLOGY, INC. [US/US]; One Computer Associates Plaza, Islandia, NY 11749 (US).

(72) Inventor: EDSON, Tully; c/o Platinum Technology, Inc., One Computer Associates Plaza, Islandia, NY 11749 (US).

(74) Agent: PERKINS, Jefferson; Piper Marbury Rudnick & Wolfe, P.O. Box 64807, Chicago, IL 60664-0807 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *Without international search report and to be republished upon receipt of that report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

WO 01/31439 A2

(54) Title: METHOD AND APPARATUS FOR BUILDING A SELF-SPECIALIZING REUSABLE, GENERIC COMPONENT

(57) Abstract: The present invention is a computer implemented method for building a self-specializing, reusable generic software component where the generic component is a control interfaced with an application for displaying text and graphics. The generic controls are of a kind similar to Microsoft Corporation's ActiveX™ controls that are used to create complex components that can be used in web-pages or other applications. Scripting extensions are loaded in a control's constructor. The control is added to the loaded list of scripting extensions as a server thereby enabling the scripting extension to drive the control. Each control has one or more associated controllers which are stored in a registry where they are easily accessed. The controllers are built from the scripting extensions to drive the controls and thereby achieve specific functions not able to be accomplished by the control itself.

METHOD AND APPARATUS FOR BUILDING A SELF-SPECIALIZING REUSABLE, GENERIC COMPONENT

BACKGROUND OF THE INVENTION

1. The Field of the Invention

The present invention relates to the fields of computing systems and modular programming. More specifically, the present invention is a method and apparatus for building a self-specializing, reusable generic component that can be used in a variety of applications.

2. The Relevant Art

Problems with traditional programming techniques stem from an emphasis placed on "procedural" code that often is extremely difficult to design, update and modify. Generally, small changes in conventionally programmed code can affect all elements of the code. Thus, minor changes made to the software in response to user demands can require major redesign and rewriting of entire programs.

Modular programming strategies, also sometimes referred to in the art as object-oriented strategies, which have become popular in recent years, tend to avoid these problems because modular methodologies focus on manipulating data rather than procedures; thus providing the programmer with a more intuitive approach to modeling real world problems. In addition, modules encapsulate related data and procedures so as to hide that information from the remainder of the program by allowing access to the data and procedures only through the module's interface. Hence changes to the data and or procedures of the module are relatively isolated from the remainder of the program. Modular code is therefore more

easily maintained as compared to code written using traditional methods, as changes to a modular, generic component will not affect the code in the other modules.

In addition, the inherent modular nature of generic components allows these components to be reused in different programs and applications. Programmers can develop libraries of "tried and true" components that can be used over and over again in different applications. This increases software reliability while decreasing development time, as reliable programming code may be used repeatedly. U.S. Patent No. 5,815,710 to Martin et al. and U.S. Patent No. 5,838,970 to Thomas both disclose detailed descriptions of object-oriented (modular) methodologies and the general advantages of modular code for non-specific applications.

Web page development and other similar applications often require creation of complex components for complex, active displays or functions. Each component represents a generalized portion of code that can be used to solve a number of different problems. To make the generalized component meet certain application requirements it is normally necessary to add specialized code to the component. The specialized code makes the component specific to the problem at hand that the developer is trying to solve. Many generalized components can be reused in a variety of ways and sometimes a control is used to solve the same problem in many different applications. For example, a developer might use a generic graph component that has special driver code to display data in an application, and use the same generic graph component with the identical driver code to display the same data on a web page.

Reuse of generic components multiple applications force developers to continually rewrite the same specialization code. This is particularly true where different languages are

used to build the specialized code for the generic component, thereby preventing the simple solution of copying the specialized code from one location to another. Rewriting the code in this instance is often the only solution.

Accordingly, it would be desirable to have a method that can build specialization code directly into the generic component. More specifically, it would be desirable to build specialized code into the generic components such that the generic components are no less generic and still maintain the reusability characteristics that make modular programming attractive.

SUMMARY OF THE INVENTION

The present invention is a computer implemented method for building a self-specializing, reusable generic software component where the generic component is a control interfaced with an application for displaying text and graphics. In the preferred embodiment, the generic controls are of a kind similar to Microsoft Corporation's Active X™ controls that are used to create complex components that can be used in web-pages or other applications. In one aspect, the method comprises the steps of loading scripting extensions in a control's constructor. The constructor is the code that executes at start-up and shut down of the control. The control is added to the loaded list of scripting extensions as a server thereby enabling the scripting extension to drive the control. Each control has one or more associated controllers which are stored in a registry where they are easily accessed. The controllers are the specialized code used to achieve specific functions.

In another aspect of the present invention, specialization code is built into a generic control using a controller.

In another aspect of the present invention, the controller is a self-contained software

logic component capable of communicating with a generic control to handle specific functions.

In another aspect of the present invention, the generic control is reusable in an application, whereby the only specialization code required is a mechanism that informs the generic control which controller to use.

In another aspect of the present invention, a data store is provided to store the name and the location of the controller.

In another aspect of the present invention, the controller may be remotely located in relation to the generic control, and downloaded or transmitted when needed.

In another aspect of the present invention, the generic control finds and loads the controller, making a call to the controller to activate the controller.

In another aspect of the present invention, the specialization code in the controller drives the generic control.

In another aspect of the present invention, the generic control, prior to shutting down, makes a call to the controller informing the controller of the eminent shut down.

In another aspect of the present invention, the controller, upon being notified of the control shutdown, saves any data that is related to the control that will be needed if the control is later activated.

In another aspect of the present invention, the control stores the name of the controller prior to the control shutting down.

In another aspect of the present invention, the control shuts down the controller prior to shutting itself down.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGS. 1 and 2 are flow charts detailing the method of the present invention using a generic control.

FIGS. 3 through 10 are code for implementing the method of the present invention using an ActiveX™ control according to a preferred embodiment of the invention.

FIGS. 11 and 12 are code demonstrating the method of the present invention in a practical application.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention is a self-specializing, reusable generic software control. More particularly, the present invention is a method for building specialized code into a generic control without making the control less generic. Controls are generalized modules that are useful to create complex components that can be used in web pages, C++ programs, and the like. In the preferred embodiment, the control can be of any type similar to Microsoft Corporation's Active X™ Diagram Control.

Each control in an application can be used to solve various problems and to achieve various tasks. To make a generic control accomplish a specific goal, it is necessary to add some form of "business logic" or "specialization code" to the control. The specialization code makes the generic control react in a specific fashion to accomplish the desired task. Often when generic controls are re-used in various applications, the specialization code associated with the control must be re-written to be compatible with the particular application.

To overcome this problem, the present invention builds the specialization code into the generic control without making the generic control less generic. To accomplish this, a controller concept is implemented whereby the specialization code communicates with the control and can be activated by the generic control to handle a specific specialization. Each time the generic control is used in an application, the only specialization code that is necessary is a mechanism to tell the control which controller to use. Once the control is notified as to the appropriate controller, the controller does the rest.

Application of the present invention's controller concept is explained using Microsoft Corporation's ActiveX™ controls. ActiveX™ controls can be used in stand alone applications as well as on web pages. Thus, the same Active X™ control may work with some specific data in a desktop application and make that data be visible on the web in another application.

In the preferred embodiment, the ActiveX™ control will go to a registry to load the list of possible controllers on this machine whenever a specialized function is necessary. The control itself must have a current controller property that would be listed with the control in the desktop or web-page application. The control uses the current controller property to find the name of the appropriate controller in the registry. When the controller name is found, the control obtains the corresponding Universal Resource Locator (URL) that points to a script file that represents the controller. The ActiveX control loads the script and calls a start-up function in the script. The script then drives the ActiveX control as if it were stored in the desktop or web-page application itself. When the application shuts down, the ActiveX™ control calls a shut down function in the controller and subsequently shuts down the controller followed by itself.

In the preferred embodiment, Microsoft's Active Scripting Engine ™ is used to build in a scripting capability to an Active X™ control known as the Active Diagram Control ™. Of course, other scripting engines may be used to build scripting capability into a control, the key being that the scripting extensions serve to load the scripting engine when the scripting extension is required to perform a specialized function.

A data store is needed on the machine that is using the control to store the name of the controller and the location of the controller itself. This data store could be a system registry, a database, a file on disk or some other form of locatable storage. The control must be able to find the data store and use it to load the controller. When the control gets specialized by an application, the application tells the control which specialized code is necessary. The control then goes to the data store and locates the controller. It can then retrieve the controller from wherever it is stored.

The list of controller names and locations are stored on the applicable machine at installation time. The controller itself, however, does not need to be located on the same machine. For instance, it could be located on a remote computer and downloaded when needed. Thus the location of the controller stored in the data store could refer to a local file, a uniform resource locator, or in any other path to the controller that the generic control would know how to access.

The control then loads the controller and makes a call into the controller to activate it. The specialization code in the controller drives the generic control in the same manner that it would if the controller were contained in the calling application. Before the control is shut down it needs to make another call into the controller to let the controller know that it is about to be shut down so that the controller can store any data that is needed later. The

control will then shut down the controller and thereafter shut down itself. If the control contains any data it will store the name of the controller with its own data. When the data set is later loaded from its data store, the control can retrieve the controller name and restart the necessary specialized code.

In the preferred embodiment, Microsoft's Active Scripting Engine™ is used to build in a scripting capability to an Active X™ control known as the Active Diagram Control ("ADC"). Of course, other scripting engines may be used to build scripting capability into a control to permit the scripting extension to perform specialized functions not previously available in the particular control. The scripts that drive the active diagram control are the actual "controllers." In the Active Diagram Control's constructor load, the desired scripting extensions are loaded. This, in turn, loads the Microsoft Active Scripting Engine™.

FIGS. 1 and 2 are general flow charts explaining the method of the present invention. It is first necessary to develop the scripting extensions of the controllers that will be used with the control. At block 1, the scripting extensions are loaded with the start-up code for the control. At block 2, the control is added to the scripting extension as a server. This allows the scripting extension to drive the control. At block 3, a class of controllers is built, each controller corresponding to a specific scripting extension.

Referring to FIG. 3, in the preferred embodiment, it is first necessary to develop the scripts for the controllers that will be used with the Active Diagram Control and then to load the scripting extensions in the ADC's constructor load. Referring to the code in FIG. 3, the scripting extensions are loaded by first creating an active scripting extensions object at lines 10 and 20. The newly created active object is then attached to the existing script object code at lines 30 and 40. In lines 50, 60 and 70, the ADC is added to the scripting extensions as a

server. This allows a script to drive the active diagram control whenever a particular controller is activated to accomplish a specific function.

Each control may have one or more associated controllers. It is necessary to build the class of controllers associated with a particular control so that the name of the controller specified in the property associated with a control may be used to locate the controller in a data store.

Accordingly, lines 110, 120 and 130 of FIG. 4 specify the name of the script to run with an associated controller, a logical name for the controller, and the path in the data store where the script is located and can be executed from. At line 140, the particular controller is initialized with a string.

Referring to FIG. 5, variables are declared at lines 170 and 180 and each controller is initialized by the following process: at line 190, it is necessary to find the first ";" in the string that represents the logical name of the controller. If no ";" is found, as determined at line 200, then the controller to be initialized is not valid. If the ";" is found, then the logical name of the controller is initialized at line 210 as every character to the left of the ";." All white space to the left and right of the logical controller name is eliminated at lines 260 and 270. If the string turns out to be empty, as determined at line 240, then the controller is not valid. Next at line 250, the name of the script to be ran with an associated controller is obtained. All white space is eliminated from the name at lines 260 and 270. Again, if the script name is empty, as determined at line 280, then the controller to be initialized is not valid. If the controller is valid, then the path for the script is obtained from the name of the script at line 290. At lines 300, 310 and 320, the full path is set in a temporary file, the file name in the full path is found and the path name, by itself, is stripped off the full path name and saved.

At this point, after the path is obtained and saved, it is established that the controller to be initialized is valid at line 330.

After building and initializing the controllers, it is necessary to build a registry and keys to get the controllers from the registry. Referring to FIG. 6, first it is necessary to create a container to hold the list of controllers at lines 340 and 350. The variable "key" that is used to get the controller entries from the registry is declared at lines 360. Starting at line 370, the list of controller strings is parsed whereby valid controllers are added to the list of controllers at line 430 and invalid controllers are released at line 460. First, though, a controller object is established at lines 380 and 390, the object being used to represent each controller string as the list is parsed. At lines 400 and 410, a key is built for obtaining the controller entries from the registry, and at line s420-460, each controller is built from the corresponding string received from the registry.

During start-up of the Active Diagram Control™, in the controls' constructor, the current controller and old controller variables are set to null values. Also, the variable used to identify the currently executing script is set to "0."

Referring to FIG. 7, once the Active Diagram Control™ is completely activated, then the user is prompted to pick a controller if the application utilizing the Active Diagram Control™ has not already set the controller, i.e., through an OLE automated call. At line 464, if the user did not already set the controller, a "choose controller" dialog box will pop-up on the screen allowing the user to select the active controller. At line 468, if a new diagram is activated while an existing diagram is already open, then the user is given the opportunity to change the controller. Finally, if a diagram is saved, it is necessary to record the name of the active controller. If the diagram is loaded, then the controller property of the control is set to

the saved controller that was used to originally create the diagram. Referring to FIG. 8, at line 470, the diagram begins saving; at line 480, the current controller is saved; at line 490, the diagram is loaded; and at lines 500 and 510, the previously saved controller is obtained so that the diagram may be activated with the proper controller at lines 516 and 518. Previously, in FIG. 8, a "choose controller" dialog box was activated if the user did not set the active controller through an automated OLE call. FIG. 9, in turn, is the code that generates the dialog box. At line 520, a dialog box containing a list of controllers is activated. Starting at line 530, after a controller is selected by the user, the dialog box is initialized and the list of controllers is scrolled until the chosen controller is found. The controller is obtained from the list at lines 540 and 550 and the controllers' name is added to the dialog box at line 560. The dialog box is shown at line 570, and if the user cancels the controller selection, then controller activation is aborted at line 580; otherwise the chosen active controller is set at lines 590-610.

Referring to FIG. 8, whenever a new active controller is set, the new controller obtained from the list of registered controllers and the corresponding script is run to drive the control. The code in FIG. 10 describes this process. At line 620, the scripting extensions are activated. At line 640, the new controller entered is the same as the current controller that is already loaded, then there is no need to find a new controller. If the new controller entered is different from the current controller, then the new controller is obtained from the list of registered controllers starting at lines 650 and 660. Each registered controller is obtained and compared to the newly entered controller at lines 670 and 680. At line 690, if the newly entered controller does not match a controller in the list, then there is no match and subsequent controllers in the registry are compared until a match is found. At line 700, once

a match is found, the corresponding script from the loaded scripting extensions is run at line 710. If another controller is currently executing, then starting at line 720 execution of the currently executing controller is halted. At line 730, the currently executing controller is notified that it is to be shut down. At lines 740 and 750, the currently executing controller is stopped and cleared. Once the new controller is able to start, as determine at line 760, the new controller is saved at line 770 so that it can be shut down later. Also, the new controller's corresponding script is stored so that it may be halted later. Finally, at line 790, the new controller is started.

FIG. 11 is code for an example controller that provides a string each time an icon is dropped on the screen canvas, and that uses the provided string to fill the icon's label. At line 800, the "start up" or "shutdown" of the example controller is confirmed. At lines 810, 820 and 830, if the controller is starting up, then the location or path where the controller is located is obtained, the applicable pallette filename containing the icon is appended to the path, and the pallette is thereafter loaded to the canvas. If a "shutdown" message was determined at line 800, the previously loaded pallette is closed at lines 840, 850 and 860.

FIG. 12 is the code that fills the icon's label using a string obtained from a user. The routine in FIG. 12 executes whenever an icon is added to the canvas. At line 870, the first item from the list of items in the pallette added to the canvas is obtained. If the item is determined to be an icon at line 880, then the user will input a string into an input bo at line 890, and the string is placed into the label at lines 900 through 950.

While there has been illustrated and described what are at present considered to be preferred embodiments of the present invention, it will be understood by those skilled in the art that various changes and modifications may be made, and equivalents may be substituted

for elements thereof without departing from the true scope of the invention. In addition, many modifications may be made to adapt a particular situation or material to the teachings of the invention without departing from the central scope thereof. Therefore, it is intended that this invention not be limited to the particular embodiments disclosed as the best modes contemplated for carrying out the invention, but that the invention will include all embodiments falling within the scope of the appended claims.

What Is Claimed:

1. A computer implemented method for building a self-specializing, reusable generic component where said generic component is a control interfaced with an application for displaying text and graphics, comprising the steps of:
loading scripts in said control's constructor;
adding said control to said scripts as a server such that a given script may drive said control; and
building a class of controllers associated with said control; and
storing said controllers in a registry.

2. The method of Claim 1, further comprising the step of providing a logical name for each controller in said class of controllers.

3. The method of Claim 1, further comprising the step of providing a location path for each controller in said class of controllers.

4. The method of Claim 1, further comprising the step of designating corresponding scripts for each controller in said class of controllers.

5. The method of Claim 1, further comprising the step of loading one or more controllers in said class of controllers from said registry.

6. The method of Claim 5, further comprising the step of providing said control with a property for storing the name of an active controller and comparing the name of the active controller to the one or more loaded controllers to find a matching controller.

7. The method of Claim 6, further comprising the step of following a pointer

obtained from said matching controller to a corresponding script for said matching controller and calling and executing said corresponding script of said matching controller to drive said control.

8. A computer implemented method for building and executing a self-specializing, reusable generic component where said generic component is a control interfaced with an application for displaying text and graphics, comprising the steps of:

loading scripts in said control's constructor;

adding said control to said scripts as a server such that a given script may drive said control;

building a class of controllers associated with said control;

storing said controllers in a registry;

activating said control;

loading one or more controllers from said class of controllers to an accessible memory location;

comparing a desired controller associated with said control to said loaded controllers and selecting a matching controller; and

calling and executing a script associated with said matching controller to drive said control.

9. The method of Claim 8, wherein said desired controller is provided in a property of said control.

10. The method of Claim 8, wherein said desired controller is provided by way of a user prompt.

11. The method of Claim 8, further comprising the step of using a new controller when a new control is activated simultaneously with said activated control.

12. The method of Claim 8, further comprising the step of saving data associated with the activated control and the name of the desired controller prior to shutting down the activated control.

13. The method of Claim 8, further comprising the step of re-activating said control, and said desired controller using said saved data and said saved name of said desired controller.

14. A computer implemented method for building and executing a self-specializing, reusable generic component where said generic component is a control interfaced with an application for displaying text and graphics, comprising the steps of:

loading scripts in said controls's constructor;

adding said control to said scripts as a server such that said scripts may drive said control;

building a class of controllers associated with said control;

storing said class of controllers in a registry;

activating said control;

loading one or more controllers of said class of controllers to an accessible memory location;

providing said control with a property to store the name of an active controller;

comparing the name of the controller stored by said property to said one or more f loaded controllers to select a matching controller;

following a pointer obtained from said matching controller to a script that represents said matching controller; and

calling and executing a function in said script of said matching controller to drive said control.

15. The method of Claim 14, further comprising the step of using a new controller when a new control is activated simultaneously with said activated control.

16. The method of Claim 14, further comprising the step of saving data associated with the activated control and the name of the desired controller prior to shutting down the activated control.

17. The method of Claim 14, further comprising the step of re-activating said control, and said desired controller using said saved data and said saved name of said desired controller.